IA-64 Architecture Disclosures White Paper

Robert Geva, Intel and Dale Morris, Hewlett-Packard

On February 23, 1999, Hewlett-Packard and Intel disclosed more details of the new IA-64 architecture. This paper discusses the architecture innovations that are being disclosed and focuses on the end-user benefits that the IA-64 architecture provides.

Existing architectures use various approaches to enable efficient implementation and high performance. Mostly, there has been an increasing trend towards optimizing compilers to generate efficient code that exploits the features of the underlying hardware. A compiler that can arrange for greater parallelism can get more performance from any given hardware. This means that simpler implementations can be built, and simpler microprocessors can be built with higher frequencies. Compiler writers responded to these opportunities and today's compiler technology provides many capable optimizations. However, in practice, compiler optimizations frequently encounter obstacles that may either render an optimization illegal in some cases, or not beneficial in other cases.

The architectural features detailed in this paper show how the IA-64 architecture enables compilers to be more successful then have been with existing architectures, and how at the same time, the architecture enables designing microprocessors that are powerful and fast.

1. Predication

Modern processors contain multiple, parallel functional units capable of working on separate instructions concurrently. The task of instruction scheduling, therefore, includes organizing the work to be performed such that multiple instructions can be executed in parallel, taking advantage of parallel hardware. However, this task is complicated by the fact that most application and server programs are written within a sequential model, where the programmer thinks of a processor as doing only one thing at a time, and designs their application accordingly.

One of the biggest impediments for a compiler, in attempting to expose greater instruction-level parallelism (ILP) in a program, is conditional branching. Control flow changes which depend on information that is not available at compile time break a program up into relatively small pieces and diminish the compiler's ability to reorganize the work to expose greater parallelism.

Additionally, at run time, conditional branches are difficult to execute efficiently because a processor must typically choose what instructions to fetch beyond a branch before it has had time to compute the condition for that branch. Elaborate mechanisms are designed with the goal of predicting future branch conditions. When the prediction proves incorrect, the processor must discard all the work it had done based on its incorrect guess. This represents a lost opportunity, and the performance cost is significant in modern processors and is growing, since the cost is equal to the product of the pipeline depth (which is increasing) times the pipeline width or number of concurrent functional units (which is increasing).

An approach which removes branches would therefore increase performance both by allowing compilers to exploit greater ILP, making better use of execution units, and by avoiding these branch penalties which leave the processor with nothing to do for cycles at a time.

Overview of predication

In many common computer languages, the programming paradigm is a step-by-step, sequential execution. Decisions may be made during the course of execution which determine which subsequent steps to perform. This is achieved by changing control flow to continue execution at a different step in the program.

Consider the if-then-else construct from c, for example. A boolean condition is computed, and then either the steps within the “then” clause or the steps within the “else” clause are executed, depending on the computed condition. In either case, control is passed to the code following the if-then-else.

In traditional architectures, this sort of construct is compiled into machine code using a conditional branch which changes the address from which instructions are fetched depending on the outcome of the condition computation. For example, this source code:

```
if (emp_status == ACTIVE) {
   n_active_emps++;
   total_payroll += emp_pay;
} else {
   n_inactive_emps++;
}
```

is typically transformed into a sequence such as this:

```
{
   cmp.ne    p1 = rs, ACTIVE  // compare emp_status
   (p1) br   else             // jump to else code if condition false
}
.label then
{
   add    rt = rt, rp         // sum total_payroll + emp_pay
   add    ra = ra, 1          // increment n_active_emps
   br     join                // skip over else code
}
.label else
{
   add    ri = ri, 1          // increment n_inactive_emps
}
.label join
```

As can be seen in this example, the conditional control flow divides the program into small sections of code, which hinders optimal scheduling. Also, if the branch is mispredicted, the performance cost is very large compared to the total work we’re trying to get done. Even if the branch condition is fairly predictable, the penalty may still be significant. For example, if the misprediction penalty is 5 cycles on a machine that can execute 4 instructions each cycle, the cost of misprediction is 20 instruction execution slots. If the prediction is 90% accurate, the cost of the branch is: 1 execution slot (for the branch) + 0.10 * 20 execution slots (penalty if we mispredict) = 3 slots. On average, there’s a lot of execution units sitting idle.

Predication transforms the original control flow dependency into a data dependency by allowing each of the instructions from the then and else clauses to either execute or not, depending on the outcome of the comparison. With predication, the code looks like this:

```
{
   cmp.eq    p1, p2 = rs, ACTIVE  // compare emp_status
} {
   (p1) add  rt = rt, rp          // sum total_payroll + emp_pay
   (p1) add  ra = ra, 1           // increment n_active_emps
   (p2) add  ri = ri, 1           // increment n_inactive_emps
}
```

Each of the instructions from the then and else clauses now have a “qualifying predicate” specified, and are combined into one basic block with no branches. If an instruction’s qualifying predicate is 1, it executes normally. If the qualifying predicate is 0, it does nothing.

For this simple example, the execution time on a processor with three or more add units is the same as in the non-predicated example, except there are no branches and so no possibility of a branch penalty.

Compare relations

Compares on general registers in IA-64 can test for any of the following relations:

Instruction	Compare Relation (a rel b)		
cmp.eq	equal	a == b	
cmp.ne	not equal	a != b	
cmp.lt	less than	a < b	signed
cmp.le	less than or equal	a <= b	signed
cmp.gt	greater than	a > b	signed
cmp.ge	greater than or equal	a >= b	signed
cmp.ltu	less than	a < b	unsigned
cmp.leu	less than or equal	a <= b	unsigned
cmp.gtu	greater than	a > b	unsigned
cmp.geu	greater than or equal	a >= b	unsigned

Two register values can be compared, or a register value can be compared to a signed immediate value. Additionally, the Test Bit instruction can be used to compare any selected bit in a general register to an immediate value.

Instruction	Test Relation
tbit.nz	selected bit == 1
tbit.z	selected bit == 0

Types of compares

IA-64 provides several “types” of compares. Each compare writes two separate predicate target registers. The type defines how the compare modifies its target predicates based on the computed comparison relation. We will explore the “normal” and “unconditional” types first.

When these instructions execute, they compute the compare result and then set the first predicate target to that result and the second predicate target to the complement of the compare result. Example:

```
cmp.eq     p1, p2 = r1, r2
```

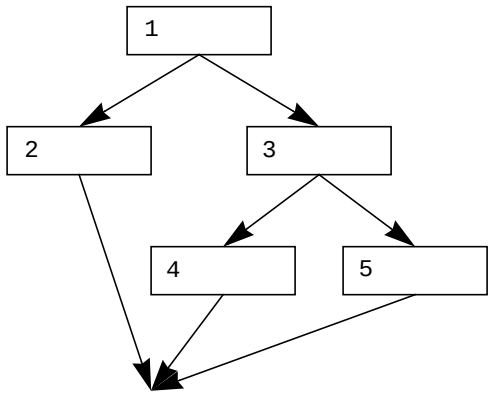
This instruction will set p1 = 1 and p2 = 0 if GR[r1] and GR [r2] are equal, and p1 = 0 and p2 = 1 otherwise. This allows for easy computation of complementary predicates for if-then-else constructs that have a simple condition. (This optimization is commonly called “if conversion”.)

Compare instructions can themselves be predicated, just like other instructions in IA-64. The difference between normal and unconditional compares is in their behavior when the compare instruction itself is predicated off.

Normal compares work as one might expect. If the qualifying predicate is 1, the compare executes as just described above. If the qualifying predicate is 0, the compare does nothing (its target registers are unchanged).

Unconditional compares are exceptional in that they write their target predicates even if the qualifying predicate is 0. If the qualifying predicate is 1, the behavior is the same as for normal compares above. But, if the qualifying predicate is 0, an unconditional compare writes 0 to both of its predicate targets. This is very useful in nested if-conversion. For example, consider this code and the corresponding control flow graph:

```
...
if (a>b) {        // block 1
   c++;           // block 2
} else {
   d += c;        // block 3
   if (e==f) {    // block 3
      g++;        // block 4
   } else {
      h--;        // block 5
   }
}
...
```



The total amount of computation here is small, and so we may wish to if-convert all of it to avoid any branch penalties and remove the barriers to other optimizations. The nested if must be coded such that neither of its then or else clauses executes if the outer else clause does not execute. The unconditional compare provides this ability The compiled code looks like this:

```
{
   cmp.gt    p1, p2 = ra, rb         // block 1
} {
   (p1) add  rc = rc, 1              // block 2
   (p2) add  rd = rd, rc             // block 3
   (p2) cmp.eq.unc  p3, p4 = re, rf  // block 3
} {
   (p3) add  rg = rg, 1              // block 4
   (p4) add  rh = rh, -1             // block 5
}
```

If (a>b), then p2 will be set to 0 by the first compare, and the unconditional compare in block 3 will force both p3 and p4 to 0, shutting off all of the code in the nested if.

Here again. one can see the benefits of predication. Although some of the code that we fetch will be predicated off, there is no downside in a parallel processor, since those functional units would have been idle anyway due to the lack of parallelism in the original sequential code. There will be no misprediction penalties, since there are no branches. And additionally, since the branches have been removed, the compiler is now free to schedule this code along with the code above and below it together, allowing more parallelism to be exposed.

Parallel compares

In addition to normal and unconditional compare types, IA-64 provides three types of parallel compare types. These are designed to enable greater parallelism and reduce critical paths in the computation of more complex comparison conditions. The three parallel types are called AND, OR, and ANDOR.

The special aspect of these types is that multiple parallel compares can target the same predicate register simultaneously. The multiple compare results are logically “anded” or “ored” into the target predicate, depending on the type. This is implemented by performing conditional writes of the predicate targets.

The AND type compares set both target predicate registers to 0 if the compare relation is false. If the compare relation is true, these compares do nothing.

The OR type compares set both target predicate registers to 1 if the compare relation is true, and otherwise do nothing.

The ANDOR compares set one target predicate to 0 and the other target predicate to 1 if the compare relation is true, and otherwise do nothing.

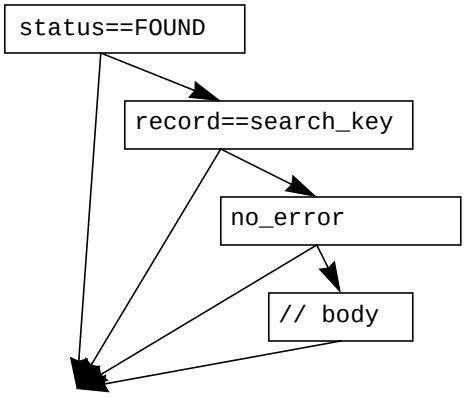
All of these parallel compares can be, themselves, predicated, and this works in the normal way. If the qualifying predicate is 0, the compare does nothing. Here’s a tabular summary of how these compares work, along with the normal and unconditional compares. A blank entry indicates that the compare does not write its target predicate:

Type	Qualifying Predicate == 1				Qualifying Predicate == 0	
	Compare Result == True		Compare Result == False			
	First Target	Second Target	First Target	Second Target	First Target	Second Target
normal	0	1	1	0		
UNC	0	1	1	0	0	0
AND	0	0				
OR			1	1		
ANDOR			1	0		

Consider this example:

```
if (status==FOUND && record==search_key && no_error) {
   ...       // body
}
```

Traditionally, this kind of construct produces this sort of control flow graph in the compiled code:



This typically then take multiple cycles to execute, with little work being done each cycle and plenty of branches to guess wrong on. With parallel compares, this sort of condition result can be computed in a single cycle on a parallel machine. An additional instruction is needed to initialize the predicates used. the code looks like this:

```
{
   cmp.eq.or     p1, p2 = r0, r0         // initialize p1 = p2 = 1
} {
   cmp.eq.and    p1, p2 = rs, FOUND      // compute(status==FOUND) &&
   cmp.eq.and    p1, p2 = rr, rk         //        (record==search_key) &&
   cmp.ne.and    p1, p2 = rn, 0          //        (no_error!=0) simultaneously
}
```

This reduces program critical path and exposes more parallelism.

Only compares of the same type may be combined to write the same predicate register simultaneously. Thus, even though multiple instructions are attempting to write the same predicate register, each will either write a 0 or not write at all (in the case of ANDs), or each will either write a 1 or not write at all (in the case of ORs).

Parallel compares use a restricted set of relations:

Instruction	Compare Relation (a rel b)		
cmp.eq	equal	a == b	
cmp.ne	not equal	a != b	
cmp.lt	less than	a < 0	signed
cmp.lt	less than	0 < b	signed
cmp.le	less than or equal	a <= 0	signed
cmp.le	less than or equal	0 <= b	signed
cmp.gt	greater than	a > 0	signed
cmp.gt	greater than	0 > b	signed
cmp.ge	greater than or equal	a >= 0	signed
cmp.ge	greater than or equal	0 >= b	signed

Predication and control speculation

IA-64's predication and control speculation capabilities work well in concert, and together enable optimizations that would not be possible with either in isolation. For example, in computing complex comparison conditions, it is often unsafe to compute subsequent a given part of the computation before previous parts have been computed. Consider this example:

```
if (p!=NULL && p->record==search_key && no_error) {
   ...       // body
}
```

Here the dereference of p is clearly unsafe to do until we've determined whether p is NULL. Control speculation features allow these sorts of computations to be done safely, deferring any exceptions and dealing with them off of the critical path of the application code.

If one or more of the register operands to a compare instruction contain a deferred exception token (NaT), the compare writes a 0 into both of its target predicate registers. This ensures that speculation failures will not cause the wrong instructions to execute. When the speculation is checked, and the recovery code is invoked, the correct computation will be completed. This behavior is summarized below:

Type	Qualifying Predicate == 1 and one or more sources have NaT	
	First Target	Second Target
normal UNC	0 0	0 0
AND OR ANDOR	0	0

2. Control Speculation

Instruction scheduling is a compiler optimization that is useful for all modern architectures and processors. In general, instruction scheduling changes the order between the instructions, so that the resulting order runs more efficiently. Typical goals of instruction scheduling include

- Balancing usage of machine resources in a given cycle
- Placing some distance between a generator of a value and a user of that value, so that the user is scheduled when the value is ready.

Obviously, the scheduler cannot reorder instructions at will, or the correctness of the program might be broken. For example,

- If an instruction uses a value, then it can not be rescheduled before the instruction that produces that value.
- An instruction should either execute under the same condition as in the original program or under different condition where the difference does not change the general behavior of the program. This means that if a program is meant to change some data value only under some specified conditions, then in general the instruction that changes that value can not be moved so that it executes regardless of these conditions.

A simple-minded implementation of a scheduler according to these criteria results in a very limited ability to move instructions around. Local schedulers are some of the simplest ones. Local schedulers partition the program into basic blocks. A basic block of instruction is the longest consecutive sequence of instructions that has one entry point and one exit point. If during the execution of a program, control flow reaches the first instruction of a basic block, then we are guaranteed that the rest of the sequence of instruction will execute in that order. Local schedulers can only move instructions within the boundaries of their original basic blocks.

Obviously, a basic block can have at most one branching instruction. In practice, in scalar codes, a branch instruction appears very frequently in the instruction stream, about once in every eight instructions or so. The small average size of a basic block, combined with the data dependence constraints imposes a great limitation on a scheduler. Some CPUs would benefit from a scheduler that can distance a generator of a value from its user by a few cycles. A local scheduler can be successful in some cases but not likely in most, because of the small number of instructions and the high level of constraints imposed by the data dependence.

Modern architectures have streamlined the instructions that processors execute to avoid long-latency operations. Memory loads, however, remain relatively long latency unless the data is already resident in the highest level cache. Further, the latency to secondary caches and main memory, expressed in processor clock cycles, continues to increase as processor clock rates go up faster than cache and memory access speeds. Also, although level 0 caches are getting larger, working set sizes are increasing at a faster rate. For these reasons, it is often beneficial to schedule loads in advance to cover longer latencies, taking advantage of parallel queuing mechanisms in non-blocking caches. So on the one hand load operations are hard to execute well and may take many cpu cycles to complete. On the other hand, efficient scheduling of load opera-

tions is key to performance, more then any other operation. The reason is that typically, a chain of computation starts with several load operations that brings values from memory to registers, then some computation is performed on the values in the registers, and finally results are stored back to memory. Since loads tend to start a chain of computation, they are first on the critical path and need to be scheduled well.

There are two main impediments to scheduling loads earlier. One is the unpredictability of dynamic control flow, and the fact that exceptions generated by a load cannot be raised until it is known that the program will actually require that load. The second main impediment is the unpredictability of address aliasing with prior store instructions. A load cannot be done in advance of a logically prior store unless it is known that the store does not modify the very memory that the load is reading. The address aliasing problem is a vexing one for compiler designers because in many situations the likelihood of address collision is extremely small, but since the compiler must ensure correctness the potential aliasing is a barrier to optimization.

IA-64 provides special mechanisms to allow the compiler to schedule loads ahead of logically prior stores and branches, even when it cannot prove that the load and store do not reference the same memory, or that it is safe to hoist the load above a branch. We call this capability “Data Speculation” and “Control Speculation”. This allows for scheduling optimizations that allow load latencies to be hidden under the execution time of other operations.

Runtime mechanisms detect collisions and provide recovery to satisfy correctness requirements.

Speculative loads, the “NAT” token.

The IA-64 architecture is designed to benefit from optimizing compilers more then traditional architectures. The architecture provides a way for the scheduler to be able to move a load instruction to other basic blocks even if the compiler can not prove that the move is safe. To accomplish that, the architecture introduces speculative loads. Speculative loads break the link between the function of the load operation that brings in a value into a register, and detection of a possible exception. In IA-64, when a speculative load executes, if the memory access is successful then the value is brought into the specified register just like a regular load would have done. If the memory access fails such that a regular load would have raised an exception, then a speculative load sets a “deferral token” in the target register. The handling of the exceptional state is deferred for a subsequent instruction. Deferral of exception means that the exception is not handled at the time it occurs. Instead, the fact that an exception occurred is noted, and handling it will happen later. The compiler is responsible to arrange for a later examination of the deferral token, and if necessary, execute a related recovery code.

The target register of a load operation can be either a general purpose register or a floating point register. The implementation of the “deferred exception token” in floating point register is a special encoding. The IEEE encoding of floating point quantities is such that there are many bit patterns that do not represent numbers. Such bit patterns are termed “not a number”. One of these bit patters is set aside in IA-64 to represent the deferred exception token. This bit pattern is termed “NatVal”. Integer quantities do not have an analogous redundancy. The implementation of the speculation token for integer registers is an extra bit in each register, so integer registers have 65 bits. The additional bit is called the “NaT” (Not a Thing) bit.

NAT Propagation and recovery code

A speculative load that encounters exceptional conditions sets the NAT token, either the 65’th bit for an integer load or the NatVal value for a floating point load. Arithmetic operations propagate the NAT token. As an example, an “add” operation of two integer registers also performs a logical “or” operation on the NAT bits of the two operands. If either one of the NAT bit is set then the “add” operation sets the NAT bit in the destination register.

When the compiler moves a load instruction from its “home” block to another block, it has two possibilities:

1. It can successfully prove that the load can execute safely in the new location, or
2. It changes the load to a speculative load, and inserts a new instruction, a speculation check, in the original location of the load.

The speculation check is a special IA-64 instruction. It takes as operand a register and an address. It checks the NaT token of the register. If the NaT token is set then control is transferred to the location specified in the second operand. If the NAT token is clear then no operation is performed. The check instruction does not consume execution cycles. Subsequent instructions that use the value in the target register do not have to wait for the check instruction: they can be executed in the same cycle.

The address given to the speculation check instruction as a second operand is the starting address of “recovery code”. It is the compiler’s responsibility to generate a sequence of code that will execute if the load fails. That recovery code will reexecute the load instruction and any dependent instruction that was also executed speculatively and used the wrong data. The recovery code will usually end with a branch back to the original location of the load, so that the program resumes execution with a correct state.

Instructions whose destination is not an integer or floating point register can not propagate the NaT token. Such instructions are store instruction, whose destination is in memory, and instructions that set the branch registers. It is the compiler’s responsibility not to move these instructions to speculative locations. Compare instructions are a special case. They set values in predicate registers, which do not have a way to represent the deferral token. Compare instructions do propagate the deferral token by setting their targets to “false”. No instruction that is predicated on the result of that compare will execute. In particular, if a branch is predicated on that predicate then it can not be taken.

Suppose a segment of code contained two load instructions and an add instruction that add the two loaded values. The compiler may speculate all of these three operations. If either of the two loads fail, and the NaT bit of its target register is set, then the target register of the add instruction will also be set by the NaT propagation. Therefore, the compiler does not have to generate two speculation check instructions, one corresponding to each one of the speculated loads. Instead, it is sufficient to generate only one speculation check instruction, and set it to check the NaT bit of the register that is the destination of the add instruction. This way, we save one speculation check instruction in the main line code, and one recovery block.

```
(p1) br.cond label
ld8 r1 = [r9]
ld8 r2 = [r8]
add R3 = r1, r2
```

Original Order

```
ld8.s r1 = [r9]
ld8.s r2 = [r8]
add r3 = r1, r2
(p1) br.cond label
chk.s r3, recover
```

Optimized order

Speculative loads and checks are efficient

Assume that a branch guards a load operation. The simplest way for a branch to guard a load from illegal memory access is if the branch is following a compare operation, that compares the value in a pointer to zero. If the value is zero then the branch is taken, and the load does not execute. If the value is non zero, then the load will execute and access memory according to the value in the pointer. For performance reasons, the compiler may want to schedule the load earlier then its original location, in particular before the branch. If a regular load executes before a branch, it might fail causing a fatal exception in the program execution. A speculative load will bring in the data if it is valid, satisfying the need to bring in the data early. If the load should not have executed, then the fatal exception will not happen. The speculative load will just set the NaT token. However, this also means that the load should not have executed, and the guarding branch should have been taken. But nothing has changed with respect to the branch instruction! It will still be taken. The check instruction, that was placed by the compiler in the original location of the load instruction, will not execute. Therefore, in a correct program, even if a load executed speculatively, and had a wrong memory reference, no penalty occurs and no recovery code executes. On the other hand, if the load should have executed, then now it executes earlier and its data is ready on time for the user of its value. Control will reach the check instruction, but the data is valid, and the check instruction does not trigger the recovery code, i.e., it’s for free!

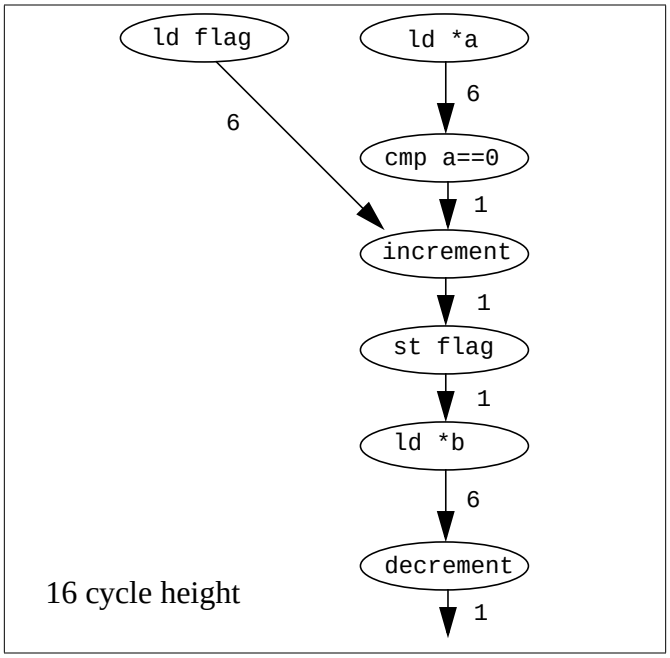
3. Data Speculation

In scheduling a load instruction early enough to cover some anticipated latency, the compiler often finds it must move the load across a store instruction. Sometimes it is possible for the compiler to prove that the load and store cannot reference the same memory, which means that such an optimization is known to be safe. For example, if both the load and the store are statically bound to two separate variables in the local stack frame, then it can know that their addresses will not overlap.

However, often the addresses for the load, or the store, or both are done through pointers. Frequently, this means that the compiler cannot determine whether the addresses might overlap. The information to make this determination may not be present within the current module being compiled, or the proof may be computationally intractable. Yet, based on common coding practices, many of these situations will rarely, if ever, result in the load and store referencing the same memory. Take the following code as an example:

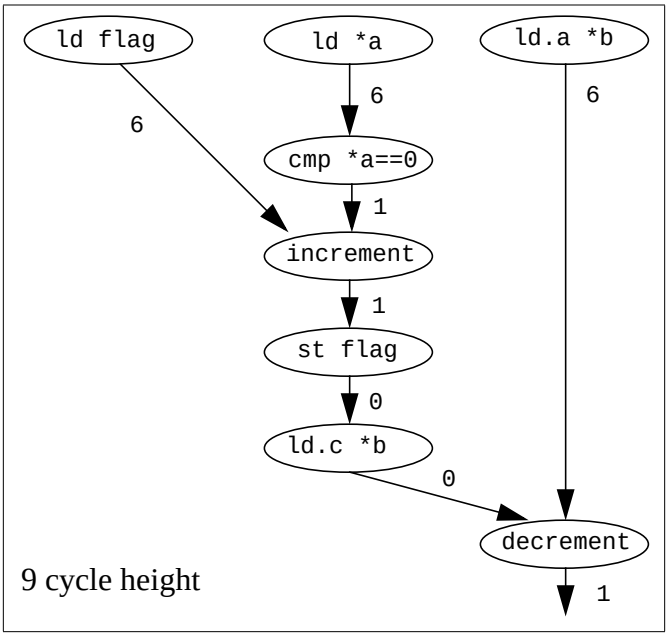
```
unsigned char    flag;               // a global

int test (int *a, *b)
{
    if (*a)
        flag += 1;
    return(*b - 1);
}
```

Assume, for this example, that we would like to schedule the loads of *a, *b and flag to cover a cache latency of 6 cycles. The dependence graph for this function is tall and narrow as shown below. We can start by loading *a and flag, but we cannot start loading *b until after first storing the updated flag, since *b may point to flag. This is unlikely, given that flag is a global, and given the differing types (int vs. char), but it is nevertheless possible, and so traditional compilers must honor this potential dependency and schedule the code in a serial fashion.



In IA-64, this possible aliasing dependency is broken by converting a load to an “advanced load” or ld.a, and then adding a “check load” instruction or ld.c at the original location in the code of the load. The ld.a performs the load “data speculatively”, and the check load tests to see if the store conflicted with it. If not, then the speculation was successful, the check load does nothing else and execution proceeds. If there was a conflict, the check load re-loads the value to give the correct execution.

This collapses the previous dependence graph to the following one:



Now that the unlikely aliasing dependency between the store of flag and the load of *b is broken, the critical path through the code is much shorter.

A compilation of this code appears below:

```
{
    ld1       rf = [radd_f]       // ld flag
    ld8       ra = [radd_a]       // ld *a
    ld8.a     rb = [radd_b]       // ld.a *b
} {
    cmp.eq    p1 = ra,0           // cmp *a==0
} {
    (p1) add  rf = rf, 1          // increment flag if *a==0
} {
    (p1) st1  [radd_f] = rf       // store new value of flag
    ld.c      rb = [radd_b]       // reload *b if speculation failed
    add       ret0 = rb, 1        // decrement
    br.ret
}
```

Conflict detection and the ALAT

In IA-64, when an advanced load is executed, it inserts an entry into the Advanced Load Address Table, or ALAT. This ALAT is the mechanism for detecting address collisions between advanced loads and stores. The inserted ALAT entry contains three pieces of information: the target register number for the ld.a, a portion of the physical address being loaded, and a size, in bytes, of the quantity being loaded.

Structure of an ALAT Entry

reg #	address	size

When the ld.a inserts the entry into the ALAT, it first checks for any existing entry with the same register number, and overwrites it, if found. Thus there can be at most one entry in the ALAT for any given register. This entry is used to detect collisions with stores and to then determine whether any collisions have occurred.

When store instructions are executed, the physical address is passed to the ALAT, which compares it with every entry in the table. Any entry which matches is removed from the ALAT. The address matching takes into account the size and starting address of the store, the address and size from each ALAT entry; and any entry which has any overlap between the two is removed.

Now consider what happens when we reach the check load. Since we are checking a specific advanced load to determine whether it succeeded, and since the check load will reload the data if the data speculation was unsuccessful, the normal usage is for the check load to target the same register as did the advanced load. When the check load executes, it first checks the ALAT to see if it contains an entry for the specified register number. If it does contain such an entry, then the speculation was successful. If one of the stores between the ld.a and the ld.c had modified the same bytes in memory which the ld.a loaded, that store would have removed the ALAT entry. Since the entry is still there, there must have been no collision. In this case, the ld.c does nothing else (e.g., it does not write its target register), and execution continues.

If the check load discovers that there is no ALAT entry for the specified register number, then there may have been a collision. In this case, the check load acts as a normal load — it reloads the value and writes it into the target register. Any subsequent instruction which reads that register must wait until the check load completes.

Essentially then, data speculation allows the load to be performed much earlier, and in situations where the load and store actually conflict, the performance simply degrades to what it would have been if the load had not been advanced.

Data speculation with hoisted uses

Although loads are among the longest latency single operations, it often occurs that the compiler would like to schedule an entire calculation which begins with one or more loads above a store which may potentially conflict. IA-64 leverages off of the mechanisms which support control speculation recovery to allow for data speculation of entire calculations.

The usage here is very similar to the simple case already described. A load is scheduled above a logically prior store, and additionally, any number of instructions which compute upon the load result are scheduled before the store as well. Collisions between the advanced load and the store are detected by the ALAT as before.

The difference for this case is in how the ALAT is checked. Rather than using a check load, which checks the ALAT and then simply reloads the data if the entry is not found, the compiler places a check instruction (chk.a) at the original location in the code of the load. This chk.a instruction is similar to a control speculation check (chk.s), except that it tests for an entry in the ALAT. The chk.a includes a register specifier, which will typically be used to specify the same register as was targeted by the advanced load. The chk.a checks the ALAT to see if it contains an entry with that register number. If it does, the check instruction does nothing else, and execution continues.

The check instruction also specifies a memory address, which is the starting point for the “recovery code”. If the ALAT does not contain the desired entry, the check instruction causes control to be transferred to the new address. That recovery code will typically re-execute the load and the dependent instructions and then branch back to the main sequence.

This allows entire computations to be scheduled independent of possible address aliasing. When the prediction of the low probability of conflict is correct, this results in higher performance. And when conflicts do occasionally occur, the proper instructions are executed to give the correct result.

ALAT Management

The design of the ALAT in IA-64 is such that no additional state need be saved on context switch. This is important in enabling implementation flexibility in the size of address matches and the number of ALAT entries. Instead, the ALAT is simply cleared of all entries on context switch.

This means that, upon re-dispatching a process which had data speculative computations in flight, those computations will be forced to execute their recovery code. This is a small penalty on switch — smaller, for example, than the cold cache effects of bringing the process’ context back into the upper level caches.

The compiler can also provide hints to hardware to indicate when particular ALAT entries are no longer needed. This allows hardware to free up these entries, avoiding the displacement of an active entry on some subsequent advanced load.

Check load and check instructions can specify that, after the specified ALAT entry is checked, it may be removed. This is done with ld.c.clr and chk.a.clr instructions. Other instructions can be used to explicitly remove all entries from the ALAT, for use in context switches, for example.

Combining control and data speculation

Often in scheduling loads, the compiler finds that it is hindered both by possibly conflicting stores and unpredictable con-

trol flow. IA-64 provides the means to marry control and data speculation with “Speculative Advanced Loads”, or ld.sa. These use the ALAT for detection of data speculation conflicts, and also defer any exceptions.

The usage is just as with normal data speculation. A ld.sa is scheduled early. Any number of instruction to compute results based on the speculatively loaded value are scheduled. And in the original place in the program for the load, the compiler schedules a chk.a.

For ld.sa, if the load detects a deferred exception, then it removes any stale entries from the ALAT for the given target register, and then does not insert an entry. This means that the chk.a will invoke recovery code if either: the load had a deferred exception, or the load conflicted with a subsequent store.

The combination of these features in IA-64 provides powerful scheduling flexibility for hiding load latency. And this advantage will only grow over time, as processor clock rates increase.

Compiler support for speculation

The purpose of supporting speculation in the architecture is to enable the compiler to be as aggressive as possible in instruction scheduling.

If an architecture allows only load operations to be hoisted, then not only is it not cost effective to build an aggressive scheduler, it is also harmful. An aggressive scheduler may end up hurting performance. The scheduler is likely to move all load operation to the beginning of the program (or of the scheduling region) leave the arithmetic operations in the middle and sink all the store operations in the end. For most CPUs this schedule clogs the load unit while the other units are idle, then clogs the arithmetic unit while the other units are idle, and finally clogs the store unit while the other units are idle. To fix it, schedulers for an architecture that only allows loads to be moved are throttled not to be too aggressive.

Many traditional architectures advocate building cpus that are simple, do not perform too complicated operation in the hardware, to facilitate higher clock frequencies. The simple hardware is supposed to be complemented by an aggressive optimizing compiler. These architectures come short in that an aggressive scheduler can not accomplish its goals, because of the limitations of branches and stores on instruction scheduling.

The combination of control and data speculation of loads as well as uses, makes it cost effective to build a compiler around the assumption that in general, most instructions can be moved from anywhere to anywhere else. It is worthwhile to build all the machinery to facilitate all possible kinds of code motion. IA-64 solves the problems traditional architectures have, and enables the compiler to deliver the performance.

4. Branch Architecture

The execution of a computer program is usually viewed as a chain of operation that conceptually consists of fetching an instruction, decoding the instruction, executing it according to its semantics, and writing back the results. Then execution proceeds to the next instructions. A branch instruction instructs the CPU to fetch the next instruction from another location, rather then the next sequential location.

Just changing the value in the instruction pointer to a new address that is provided in the branch instruction is simple. That is the case with IP relative, unconditional branches. However, most branches require some processing before this simple operation can be done. Conditional branches require the processor to evaluate a given condition, and either change the value in the instruction pointer or not, depending on the result of the evaluation. Indirect branches have the new instruction pointer value stored in either a register or in memory; the processor has to fetch these values as part of executing the branch instruction. Waiting for the outcome of either of these operations delays the processor

Many desktop and server applications are characterized by irregular execution patterns, governed by many mispredicted branches. To optimize execution of scalar codes that are found in desktop and in server applications, the architecture has to address the execution of branch instructions.

IA-64 branch architecture

Predication is an architectural feature that allows the compiler to minimize the use of branches. The branch architecture complements predication by enabling efficient execution of branches.

Branching instructions

The IA-64 instruction set defines two basic branch formats.

1. In the IP relative branch format, the instruction pointer is incremented by an offset which is provided in the encoding of the branch instruction. The offset is a signed 21 bit value. This range allows branching within ±16 Mbytes of the current instruction pointer.
2. In the indirect branch form, a value is assigned to the instruction pointer from a branch register. The architecture has defined eight 64 bit branch registers.

Branches can be predicated in the exact same manner as the rest of the IA-64 instructions. A previously executed compare instruction sets a value in a predicate register. The execution of the branch depends upon whether the value in that predicate register is true or false.

The IA-64 architecture allows the compare and its dependent branch to be scheduled in the same instruction group and execute in the same cycle.

Multi-way branches

The IA-64 architecture does not require branch instructions to be the last instruction in an instruction group. If an instruction group has instructions after a branch instruction, then those will execute if the branch is not taken. In particular, the architecture provides templates that allow more than one branch among its three instructions. These templates are called multi-way branches. Instructions in bundles are sequenced, and the order of evaluation is from slot 0 in the bundle to slot 2. The implication on multi-way branches is that if an earlier branch is taken then the following ones are not executed. Combined with the architectural ability to schedule compares and dependent branches in the same cycle, multi-way branches provide a powerful way to implement high language multi-way branches such as switch statements in C.

Instruction scheduling sometimes provides another opportunity to use multi-way branches. If the scheduler can move all instructions that originate in some basic block to other basic blocks, and if originally that basic block ended with a branch instruction, then after the scheduling process only the branch instruction will be left in that basic block. The compiler can then combine that branch with the last bundle of the preceding basic block, instead of having to bundle the branch instruction as a single instruction- this improves performance and saves code size.

Multi-way branches eliminate the need for “fully qualified” predicates

If an instruction group has a number of predicated branches, the sequential semantics imply that the first one will be evaluated first, the second one will be evaluated only if the first one is not taken, and so on. Assume the first branch is predicated by predicate register P1, the second one by P2 and so on. Then the sequential semantics imply that the second branch will be taken if P2 evaluates to “true” and also P1 evaluates to “false”. Similarly, the third branch will be taken if P3 evaluates to “true”, and both P1 and P2 evaluate to “false”. This provides an optimal translation for high level language sequences of the form if (b1) then S1 else if (b2) then S2 else if (b3) then S3...

```
If (b1) then {
    S1; // large block of code
} else if (b2) then {
    s2; // large block of code
} else if (b3) then {
    s3; // large block of code
}
```

High level language code

```
.mii {
cmp.ne p1,p0 = r1, r0
cmp.ne p2,p0 = r2, r0
cmp.ne p3,p0 = r3, r0
} .bbb {
(p1) br addr_of_s1
(p2) br addr_of_s2
(p3) br addr_of_s3
}
```

IA-64 code with a multi-way branch

The high level language semantics require S2 to execute if b1 is “false” and b2 is “true”. The translation does not have to generate a predicate that corresponds to the negation of b1 “anded” with b2. The sequential semantics allow the compiler to generate fewer instructions to set the correct predicate register, resulting in more efficient code.

An alternative semantic definition might have stated that branches in a multi-way branch can be evaluated in any order, or

all at once, and that a compiler can not rely on the order of evaluation. In that case, the compiler would have had to generate instructions to set the value in the predicate P2 to be true if b1 is zero and b2 is non zero, and similarly for the other predicate registers. This alternative way obviously requires more instructions and is therefore less efficient.

Optimizing “switch” statements.

Many integer applications, such as data base servers and compiler, make extensive use of switch statements, which are branches that can have arbitrarily many targets. Compilers typically translate switch statements into jump tables. To implement a jump table, the compiler stores a list of addresses in memory. The addresses are that possible targets of the branch. In run time, the switch condition is evaluated, and the result is added to the address of the jump table. The result of the addition is used as an operand of an indirect branch.

While jump tables are a valid and efficient way of implementing switch statements in IA-64, they can sometimes be wasteful, especially if the jump table is large and sparse. IA-64 allows an unlimited number of branch instructions in an instruction group. Therefore, an IA-64 specific translation of a switch statement can be constructed from a number of compare instructions, that compare the switch expression to all possible outcomes, and a corresponding number of branch instructions. An implementation will execute as many branch instruction in a cycle as it has branch units.

Loop closing branches

State of the art branch predictors use the history of a branch to predict its next outcome. A branch that was taken more then once or twice in a raw will always be predicted taken the next time it is encountered. Traditional predictors always mispredict the last iteration of a loop. For loops that execute thousand of iterations at a time this misprediction ratio is negligible.

However, this creates a problem in integer applications, where loops tend to execute a small number of iterations. If a loop executes eight iteration on the average, and the branch prediction mechanism mispredicts the last iteration every time, then the prediction rate for this loop closing branch is less then 87.5 percent. This is much lower then average prediction rates for industry standard branch predictors. In fact, these loop closing branches have to be regarded as poorly predicted branches!

This result is surprising at first, given the intuition that branches that implement loops are the simplest and most predictable ones

IA-64 provides special support for loop closing branches, that makes them perfectly predicted. The support consists of two items:

- A special register, called LC.
- A special instruction, CLOOP.

To use the loop support, the compiler generates an instruction that assigns LC a value that is the number of iteration the loop will to execute. Then the loop closing branch is implemented with the CLOOP instruction instead of a branch instruction. The CLOOP instruction checks the value of the LC register, If that value is zero then no action is taken, and control falls through to the next sequential instruction. If the value in LC is non zero then it is decremented and control is passed back to the beginning of the loop.

5. Register Rotation in IA-64

IA-64 includes a register renaming mechanism which causes a subset of registers to appear to rotate. This renaming applies to the General Registers (GRs), Floating-point Registers (FRs), and Predicate Registers (PRs).

This register rotation allows for loop optimizations which reduce loop overhead and increase effective parallelism, thus increasing performance. These optimizations can be applied both to “counted” loops (loops where the trip count can be computed prior to beginning the loop) and data-terminated or “while” loops (loops where the trip count is dependent on calculations performed inside the loop).

In addition to the rotation mechanism, IA-64 includes two loop-related Application Registers (the Loop Count register, or LC, and the Epilog Count register, or EC) and a set of special loop branches. Together, these features enable a type of loop optimization termed “modulo scheduling”.

This section describes these features and how they are used in modulo-scheduling code optimizations.

Rotation mechanism and RRB

Each of the three register sets, GRs, FRs and PRs, are divided into two subsets: static and rotating, as shown below:

Register set	Static subset	Rotating subset
General Registers	GR 0 - 31	GR 32 - 127
Floating-point Registers	FR 0 - 31	FR 32 - 127
Predicate Registers	PR 0 - 15	PR 16 - 63

A special register holds the register rotation base, or RRB. This RRB is used to rename the registers within the rotating subset of each register set. When an IA-64 instruction references a GR x, for example, the physical register accessed is determined as follows:

- If (x<32), access physical GR [x]
- If (x≥32), access physical GR [x + RRB]

Special loop branches cause the RRB to be decremented. This has the effect of making it appear to software as if the registers have moved up one register position.

Example: suppose that RRB is 5 and that an instruction writes the value 0x1234 into GR 37. The renaming mechanism calculates this to be physical register 37 + 5 = 42. Now say that a loop branch is executed which decrements RRB to 4. At this point, if software wishes to refer to the same physical register, it accesses GR 38 (38 + 4 = physical register 42).

The rotating subset for each register set wraps around, so the addition described above is done in modulo-arithmetic, as shown below:

Before Rotation	
GR Number	Contents
127	0x7f
126	0x7e
...	
34	0x22
33	0x21
32	0x20
31	0x1f
...	
0	0x0

After Rotation	
GR Number	Contents
127	0x7e
126	0x7d
...	
34	0x21
33	0x20
32	0x7f
31	0x1f
...	
0	0x0

Loop support

It is important for performance to optimize loops well, both in parallel and in scalar code. The loop features in IA-64 allow for optimization of both counted and while loops. These optimizations increase efficiency and parallelism, enabling the better use of larger numbers of functional units.

Because these optimizations involve little overhead and little code expansion, they can be used on loops with small iteration counts as well as large trip-count loops. This means that while loops can be optimized even if little is known about the likely iteration count.

A simple example

For something simple, we'll take a counted loop with a trip count known at loop entry time.

```
for (i=0; i<n; i++) {
    *b++ = *a++;
}   /* copy string */
```

The computation in this loop is pretty simple: a load, a store and two adds. For this example, we'll say that we're scheduling for a machine with 2 load/store pipes, 2 integer execution units, a 1-cycle add latency, and a 1-cycle load latency to L0 cache. Further, we'll assume that we're scheduling assuming that a[] will be in L0. A simple, non-pipelined schedule would look like this:

Simple Schedule

ld1		
	st1	br.cloop1

The execution of this schedule would give this sequence:

Simple Loop Execution Sequence

Cycle			
1	ld1		
2		st1	br.cloop1
3	ld2		
4		st2	br.cloop2
5	ld2		
6		st2	br.cloop2

And the code would look like this:

```
// setup ra,rb,LC, check n>0
.label loop
{
    ld8    rt = [ra],8
} {
    st8    [rb] = rt,8
    br.cloop  #loop
}
```

The special br.cloop instruction automatically compares LC to 0, decrements LC and loops back.

This is a simple, non-overlapping memory copy. The throughput is 2 cycles/word, and in those 2 cycles we execute 3 instructions.

Modulo-scheduling

Modulo-scheduling allows multiple iterations of the loop to be overlapped, increasing the parallelism. New iterations are begun before previous iterations have finished. Multiple registers are used to represent a single logical variable in the original source loop, which provides a kind of pipelining of operations which have multiple-cycle latencies. In typical architectures, this use of registers requires extra instructions in the loop body, and has the downside of bloating the code size. This has the effect of increasing performance, but only for large trip-count loops. Small trip-count loops can typically be slowed down significantly, making it a hazardous optimization for loops with trip-counts unknown at compile time.

The features in IA-64 allow for efficient modulo-scheduling of loops with very little overhead, allowing the optimization to be performed much more frequently.

The execution sequence for a modulo-scheduled loop looks like this:

Modulo-Scheduled Loop Execution Sequence

Cycle			
1	ld1		
2	ld2	st1	br.ctop1
3	ld3	st2	br.ctop2
4	ld4	st3	br.ctop3
5	ld5	st4	br.ctop4
6		st5	

The schedule for this loop is of this form:

Schedule of a Modulo-Scheduled Loop

ld1			Prologue
ld2	st1	br.ctop	Kernel
	st2		Epilogue

The prologue code fills our software “pipeline”. In the kernel of the loop, we are starting and finishing one source iteration each kernel iteration. For iteration counts greater than 2, the kernel portion of the schedule is simply repeated. In the epilogue, we finish up the remaining work for the last iteration, essentially draining our “pipeline”.

The code looks like this:

```
// setup ra,rb,LC, check n>0
{
   ld8    r33 = [ra],8
}
.label loop
{
   ld8    r32 = [ra],8
   st8    [rb] = r33,8
   br.ctop   #loop
} {
   st8    [rb] = r33,8
}
```

The throughput is now 1 cycle/word, with 3 instructions executed each cycle. Note that the register rotation mechanism provides the renaming for the intermediate results in flight from previous iterations so that no register copy operations are required in the body of the loop.

Rotating predicates

The combination of predication with register rotation within the predicate register set provides for further efficiency by allowing the loop to be modulo scheduled without requiring separate prologue and epilogue code.

Rotating predicates allow us to fold the prologue and epilogue code into the kernel, turning off instructions which do not correspond to real source loop iterations. This reduces code size.

We assign predicates to the instructions in the loop. The schedule, then, looks like this:

Schedule of a Predicated Modulo-Scheduled Loop

(p16) ld	(p17) st	br.ctop	Prologue / Kernel / Epilogue

Essentially, what this does is to construct a pipeline of predicate values parallel to the pipeline of data values in the GRs. These predicates indicate whether particular instructions correspond to real source iterations, or not. Our software pipeline looks like this:

Software Pipeline

	Predicate		
Cycle	p16	p17	
1	ld		
2		st	br.ctop

The code looks like this:

```
// setup ra,rb,LC,EC,p16,p17 check n>0
.label loop
{
   (p16)  ld8    r32 = [ra],8
   (p17)  st8    [rb] = r33,8
   br.ctop   #loop
}
```

To prime the pipeline, p16 is initialized to 1 and p17 to 0. This shuts off the store in the first iteration, since it does not correspond to any source iteration, allowing us to effectively execute the prologue using the kernel code. When the loop is done, and we are draining the pipeline, p16 will be 0 and p17 will be 1, causing just the store to execute. This effectively allows us to use the kernel code to execute the epilogue code.

How this pipeline of predicates allows us to fold the prologue and epilogue into the kernel may still be a little fuzzy. The next section will make this clear.

Epilogue count and branch semantics

Because of the software pipeline depth of 2, we will need to go around the above loop n+1 times: n times to start the n source iterations, plus one more time to finish the last iteration. The execution sequence we want is shown here:

Predicated Modulo-Scheduled Loop Execution Sequence

	Stage Predicate			Predicate Value		LC	EC
Cycle	p16	p17		p16	p17		
1	ld1		br.ctop	1	0	n-1	2
2	ld2	st1	br.ctop	1	1	n-2	2
3	ld3	st2	br.ctop	1	1	n-3	2
	|	|	|	|	|	|	|
n	ldn	stn-1	br.ctop	1	1	0	2
n+1		stn	br.ctop	0	1	0	1
after loop exit				0	0	0	0

This could almost be done by simply incrementing the value we program into LC to give us an iteration count of n+1. However, although this would give us the correct trip count, it would not allow us to generate the ending predicate sequence to turn off the load in the last iteration.

Instead, IA-64 includes a second loop-related Application Register called the Epilog Count register or EC. EC is used to program the number of loop iterations required to drain the software-pipeline, after the final source iteration has been started.

IA-64 has special loop branches which use EC to determine whether to loop back or not. For br.ctop branches, each time the br.ctop is executed, LC is tested. If LC>0, LC is decremented, and the branch is taken. When LC reaches 0, then we have entered the epilogue for the loop. If LC==0, then EC is tested. If EC>1, EC is decremented and the branch is taken. When the count in both LC and EC is exhausted, the branch falls through.

The loop branches also write a predicate register. This provides the mechanism for shutting down the software pipeline during the epilogue. If LC>0, then the branch writes a 1 into p16. (Or, to be more exact, it writes 1 to the physical predicate register which will be named p16 after the rotation.) If LC==0, the branch writes a 0 into p16.

The effect of this can be seen in the above execution sequence. In the first cycle, p16 and p17 have the values 0 and 1

which they were initialized to before starting the loop. The load executes, but the store does not. In cycle 2, after executing the first br.ctop, the predicates have been rotated so that the value that was in p16 is now in p17. Additionally, the br.ctop wrote a 1 into p16.

This continues as long as we are starting new source iterations. When we reach cycle n, we have started the last source iteration and LC is now 0. The br.ctop still branches back, however, because the EC value has not been exhausted. The branch now writes a 0 into p16, giving the 0 1 combination in the last cycle. This has the effect of shutting off the load, and allowing the final store to complete.

Note that with IA-64 code, the throughput is maximized (as it was in the previous simple modulo-scheduled example) at 1 cycle/word, but the code size is the same as it was for the original simple loop.

Extrapolating

The real advantages of the IA-64 loop features become more apparent with code that is more like the real world. For the example above, the overhead of doing software pipelining without these features is fairly small, in terms of total instructions, since the loop body is very simple, and the instruction latencies are tiny. As the loop body increases, in both number of operations and latency, the amount of code growth needed to get maximum performance increases dramatically. However, with the IA-64 loop features, these optimizations scale well to larger loops without significant overhead.

As an example, let's take the previous problem, but rather than scheduling for a single cycle load latency, let's schedule for the latency of a 4-cycle higher level cache. The software pipeline, then, looks like this:

Software Pipeline

	Predicate		
Cycle	p16	p20	
1	ld		
2			
3			
4			
5		st	br.ctop

The schedule, however, is identical to the previous one, except that we use different predicates. And the code looks very similar to what we had before as well:

```
// setup ra,rb,LC,EC,p16,p17,p18,p19,p20 check n>0
.label loop
{
   (p16)  ld8    r32 = [ra],8
   (p20)  st8    [rb] = r36,8
   br.ctop   #loop
}
```

Note that all we've done is to allocate more registers for intermediate pipelined values. We also will program EC to cover the longer epilogue (now 4 cycles). The execution sequence looks like this:

Predicated Modulo-Scheduled Loop Execution Sequence

	Stage Predicate			Predicate Value					LC	EC
Cycle	p16	p20		p16	p17	p18	p19	p20		
1	ld1		br.ctop	1	0	0	0	0	n-1	5
2	ld2		br.ctop	1	1	0	0	0	n-2	5
3	ld3		br.ctop	1	1	1	0	0	n-3	5
4	ld4		br.ctop	1	1	1	1	0	n-4	5
5	ld5	st1	br.ctop	1	1	1	1	1	n-5	5
	|	|	|	|	|				|	|
n-1	ldn-1	stn-5	br.ctop	1	1	1	1	1	1	5
n	ldn	stn-4	br.ctop	1	1	1	1	1	0	5
n+1		stn-3	br.ctop	0	1	1	1	1	0	4
n+2		stn-2	br.ctop	0	0	1	1	1	0	3
n+3		stn-1	br.ctop	0	0	0	1	1	0	2
n+4		stn	br.ctop	0	0	0	0	1	0	1
after				0	0	0	0	0	0	0

Note that the throughput is still 1 cycle/word, and the code size is unchanged from the previous, short-latency example. The speedup is now 5x compared to a simple, non-overlapped loop. We now have many more intermediate values in flight in the loop, but the rotation mechanism provides the means to manage these multiple pipelined values with no increase in code size.

While Loops

These same optimizations also allow for efficient modulo-scheduling of loops whose trip count is not known at the time the loop starts executing. For such “while” loops, the loop condition is computed in the body of the loop with normal compare operations (rather than by comparing the value in LC). IA-64 includes special while-type br.wtop branches, which loop as long as a given predicate is true, and then when the branch predicate becomes false, the branch continues to loop while decrementing EC until the count is exhausted.

Throughput improvements and code size for such while loops are similar to the examples given for counted loops. Additionally, since the overhead is so small, the compiler is encouraged to accelerate such loops even when the trip count may turn out to be small. This provides a broad performance increase to loop code, which has sizeable impact on application performance.